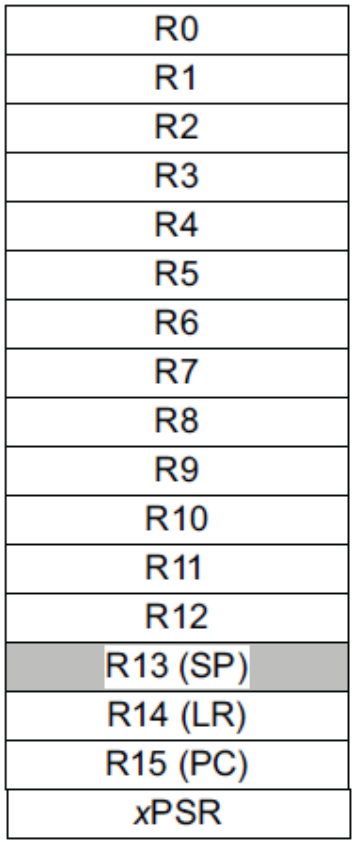
Design of Microprocessor-Based Systems

Prabal Dutta
University of Michigan

Modified by Jim Huang <jserv.tw@gmail.com>

Lecture: Architecture, Assembly, and ABI



Slides developed in part by
Mark Brehob

Review

- *What distinguishes embedded systems?*
  - Application-specific
  - Resource-constrained
  - Real-time operations
  - Physically-embodied
  - Software runs “forever”

- *Technology scaling is driving “embedded everywhere”*
  - Microprocessors
  - Memory (RAM and Flash)
  - Imagers (i.e. camera) and MEMS sensors (e.g. accelerometer)
  - Energy storage/generation

Architecture

In the context of computers,
what does architecture mean?

Architecture has many meanings

- *Computer Organization (or Microarchitecture)*
  - Control and data paths
  - Pipeline design
  - Cache design
  - ...
- *System Design (or Platform Architecture)*
  - Memory and I/O buses
  - Memory controllers
  - Direct memory access
  - ...
- *Instruction Set Architecture (ISA)*

What is an
Instruction Set Architecture (ISA)?

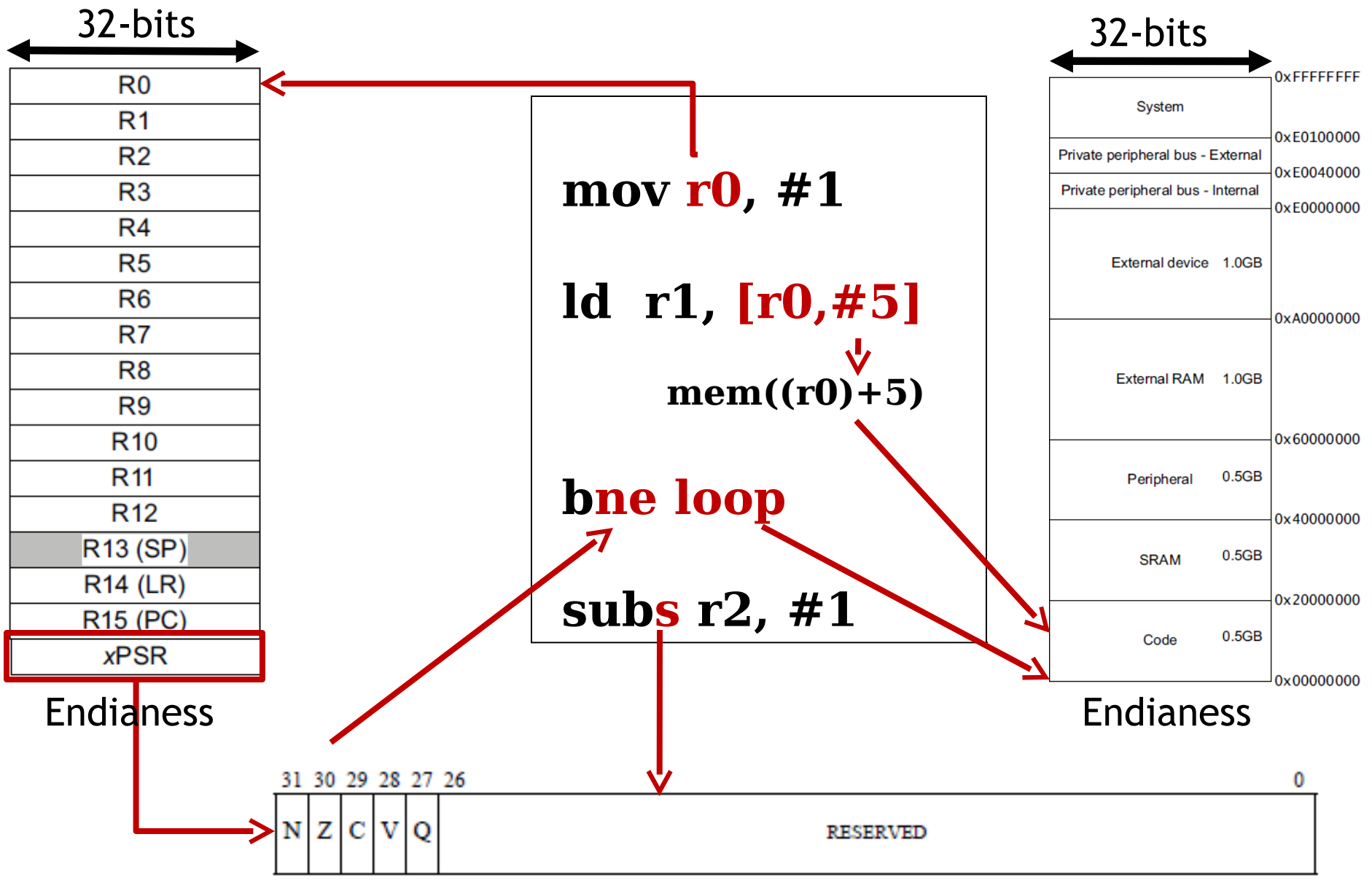
Instruction Set Architecture

"Instruction set architecture (ISA) is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine"

IBM introducing 360 in 1964

Major elements of an Instruction Set Architecture

(registers, memory, word size, endianess, conditions, instructions, addressing modes)



An ISA defines the hardware/software interface

- *A "contract" between architects and programmers*
- *Register set*
- *Instruction set*
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes
- ***Calling conventions***
  - Really not part of the ISA (usually)
  - Rather part of the ABI
  - But the ISA often provides meaningful support.

ARM Architecture roadmap

ARM7TDMI
ARM922T

Thumb instruction set

ARM926EJ-S
ARM946E-S
ARM966E-S

Improved ARM/Thumb Interworking

DSP instructions

Extensions:

Jazelle (5TEJ)

ARM1136JF-S
ARM1176JZF-S
ARM11 MPCore

SIMD Instructions

Unaligned data support

Extensions:

Thumb-2 (6T2)

TrustZone (6Z)

Multicore (6K)

Cortex-A8/R4/M3/M1

Thumb-2

Extensions:

v7A (applications) – NEON

v7R (real time) – HW Divide

V7M (microcontroller) – HW Divide and Thumb-2 only

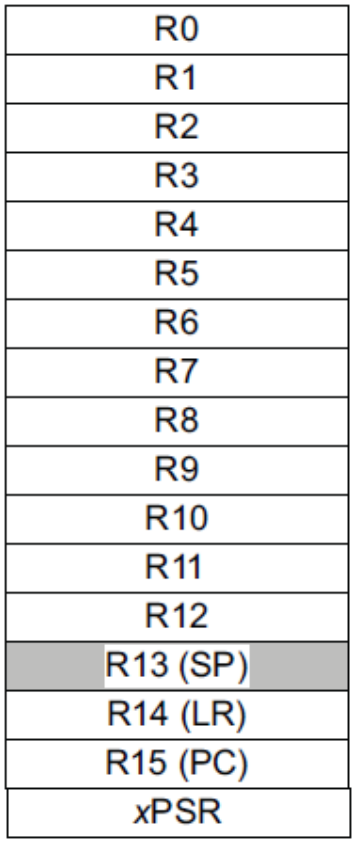
ARM Cortex-M3 ISA

Instruction Set

```
ADD Rd, Rn, <op2>
```

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

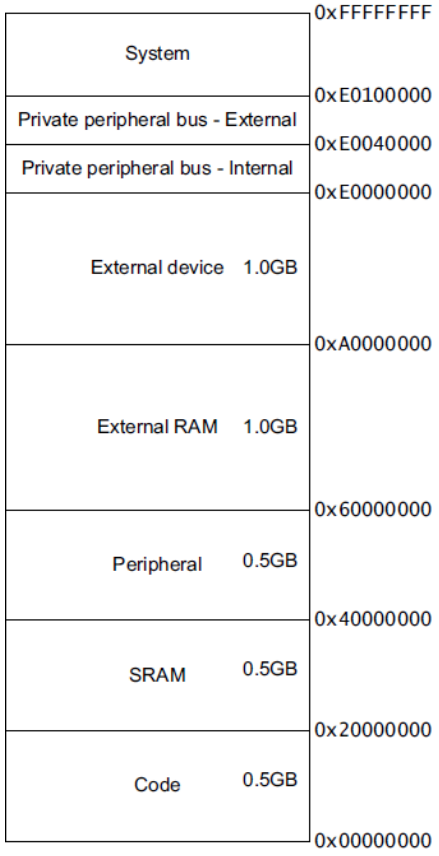
Register Set



32-bits

Endianess

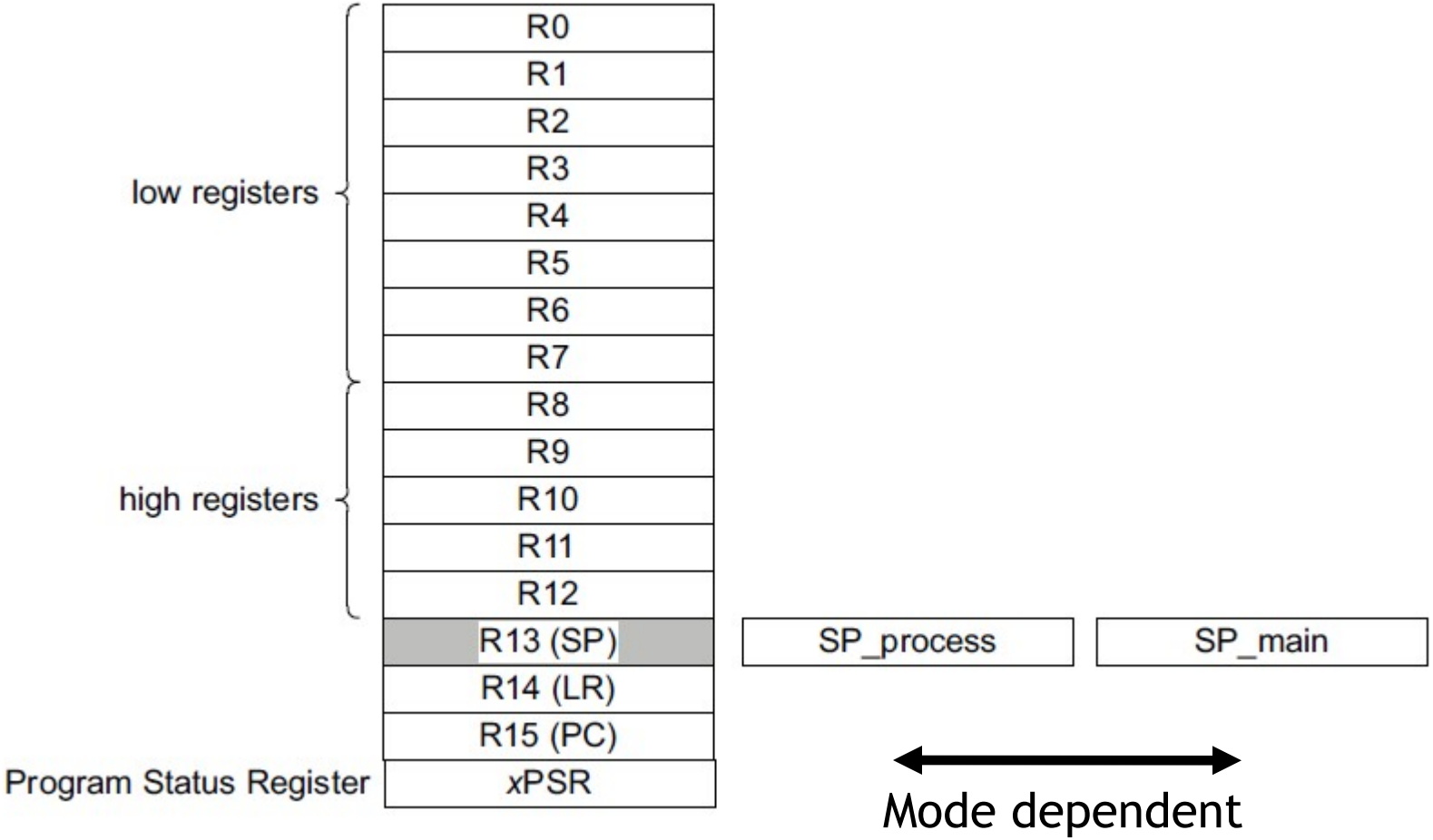
Address Space



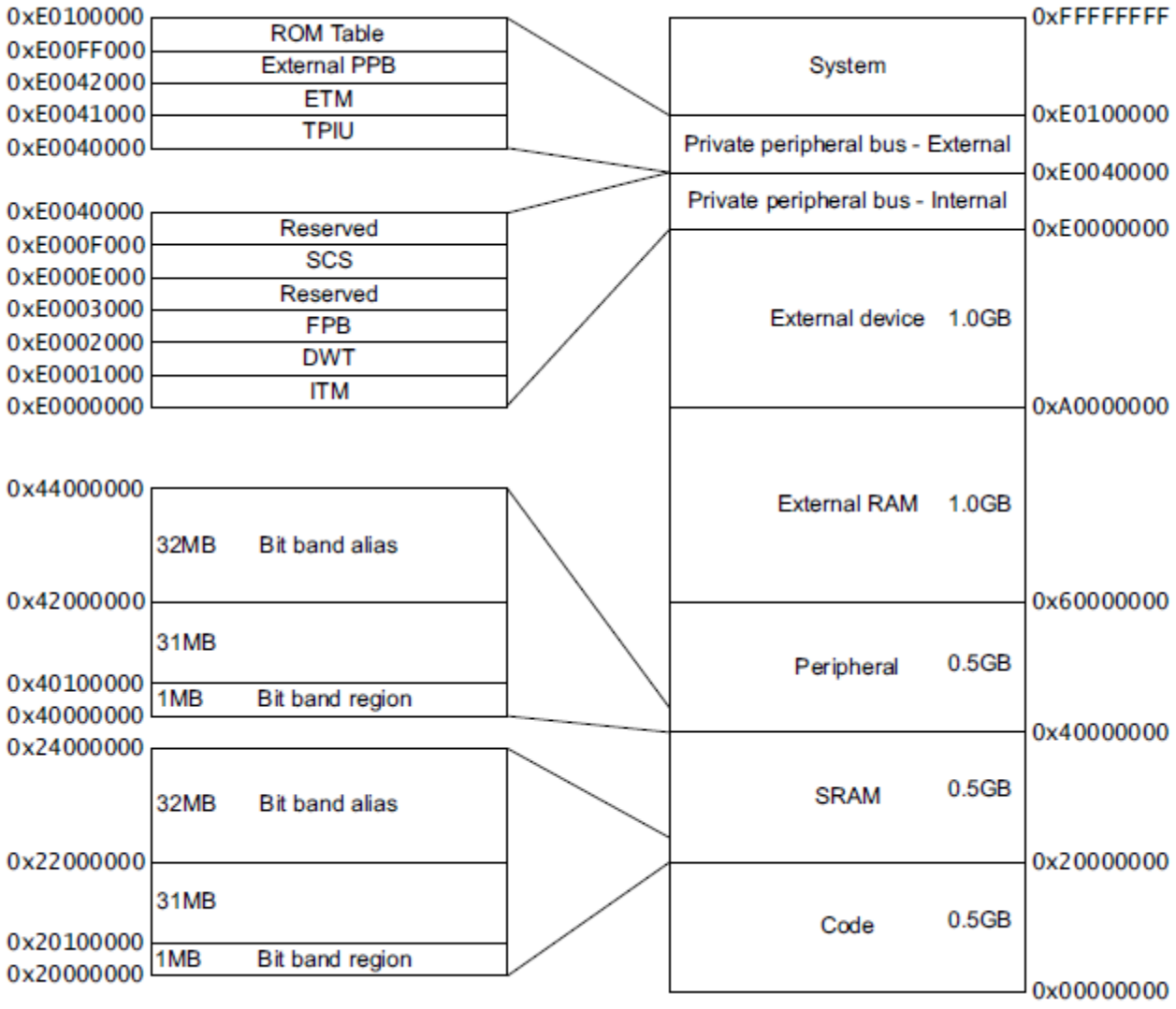
32-bits

Endianess

Registers


low registers
high registers
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
Program Status Register
xPSR
SP_process
SP_main
Mode dependent

Address Space

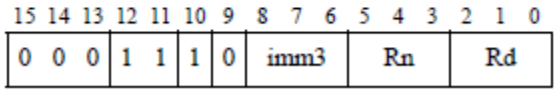


Instruction Encoding
ADD immediate

Encoding T1 All versions of the Thumb ISA.

ADDS <Rd>,<Rn>,#<imm3>

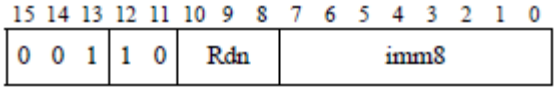
ADD<c> <Rd>,<Rn>,#<imm3>



Encoding T2 All versions of the Thumb ISA.

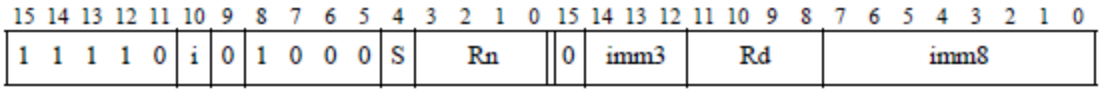
ADDS <Rdn>,#<imm8>

ADD<c> <Rdn>,#<imm8>



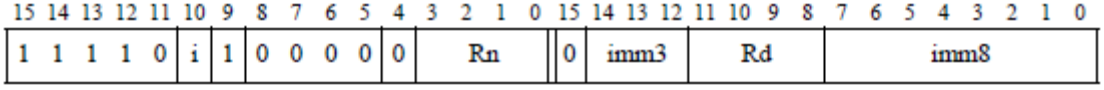
Encoding T3 ARMv7-M

ADD{S}<c>.W <Rd>,<Rn>,#<const>



Encoding T4 ARMv7-M

ADDW<c> <Rd>,<Rn>,#<imm12>



Branch

Table A4-1 Branch instructions

Instruction	Usage	Range
B on page A6-40	Branch to target address	+/–1 MB
CBNZ, CBZ on page A6-52	Compare and Branch on Nonzero, Compare and Branch on Zero	0-126 B
BL on page A6-49	Call a subroutine	+/–16 MB
BLX (register) on page A6-50	Call a subroutine, optionally change instruction set	Any
BX on page A6-51	Branch to target address, change instruction set	Any
TBB, TBH on page A6-258	Table Branch (byte offsets)	0-510 B
	Table Branch (halfword offsets)	0-131070 B

Data processing instructions

Table A4-2 Standard data-processing instructions

Mnemonic	Instruction	Notes
ADC	Add with Carry	-
ADD	Add	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
ADR	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
AND	Bitwise AND	-
BIC	Bitwise Bit Clear	-
CMN	Compare Negative	Sets flags. Like ADD but with no destination register.
CMP	Compare	Sets flags. Like SUB but with no destination register.
EOR	Bitwise Exclusive OR	-
MOV	Copies operand to destination	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See *Shift instructions* on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.

Many, Many More!

Load/Store instructions

Table A4-10 Load and store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load exclusive	Store exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
two 32-bit words	LDRD	STRD	-	-	-	-

Miscellaneous instructions

Table A4-12 Miscellaneous instructions

Instruction	See
Clear Exclusive	*CLREX* on page A6-56
Debug hint	*DBG* on page A6-67
Data Memory Barrier	*DMB* on page A6-68
Data Synchronization Barrier	*DSB* on page A6-70
Instruction Synchronization Barrier	*ISB* on page A6-76
If Then (makes following instructions conditional)	*IT* on page A6-78
No Operation	*NOP* on page A6-167
Preload Data	*PLD, PLDW (immediate)* on page A6-176
	PLD (register) on page A6-180
Preload Instruction	*PLI (immediate, literal)* on page A6-182
	PLI (register) on page A6-184
Send Event	*SEV* on page A6-212
Supervisor Call	*SVC (formerly SWI)* on page A6-252
Wait for Event	*WFE* on page A6-276
Wait for Interrupt	*WFI* on page A6-277
Yield	*YIELD* on page A6-278

Addressing Modes

- *Offset Addressing*
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- *Pre-indexed Addressing*
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- *Post-indexed Addressing*
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

<offset> options

- *An immediate constant*
  - #10
- *An index register*
  - <Rm>
- *A shifted index register*
  - <Rm>, LSL #<shift>
- *Lots of weird options…*

A5.3.2 Modified immediate constants in Thumb instructions

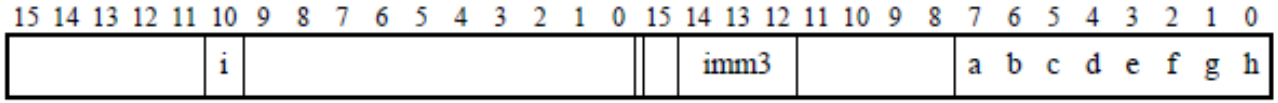


Table A5-11 shows the range of modified immediate constants available in Thumb data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

Table A5-11 Encoding of modified immediates in Thumb data-processing instructions

i:imm3:a	<const> [a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh [b]
0010x	abcdefgh 00000000 abcdefgh 00000000 [b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh [b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000
.	.
.	. 8-bit values shifted to other positions
.	.
11101	00000000 00000000 000001bc defgh000
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0

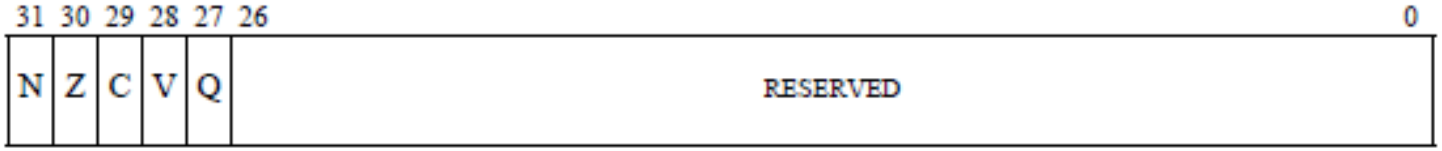
a. In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

b. UNPREDICTABLE if abcdefgh == 00000000.

ARMv7-M Architecture Reference Manual

ARMv7-M_ARM.pdf

Application Program Status Register (APSR)



APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

Updating the APSR

- *SUB Rx, Ry*
  - Rx = Rx - Ry
  - APSR unchanged
- *SUBS*
  - Rx = Rx - Ry
  - APSR N, Z, C, V updated
- *ADD Rx, Ry*
  - Rx = Rx + Ry
  - APSR unchanged
- *ADDS*
  - Rx = Rx + Ry
  - APSR N, Z, C, V updated

Conditional execution:
Append to many instructions for conditional execution

Table A6-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [ab]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [c]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [d]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [e]	Always (unconditional)	Always (unconditional)	Any

Exercise:
What is the value of r2 at done?

```
…
start:
        movs r0, #1
        movs r1, #1
        movs r2, #1
        sub  r0, r1
        bne  done
        movs r2, #2
done:
        b    done
…
```

Solution:
what is the value of r2 at done?

```
...
start:
        movs r0, #1       // r0 ← 1, Z=0
        movs r1, #1       // r1 ← 1, Z=0
        movs r2, #1       // r2 ← 1, Z=0
        sub  r0, r1 // r0 ← r0-r1
                          // but Z flag untouched
                          // since sub vs subs
        bne  done         // NE true when Z==0
                          // So, take the branch
        movs r2, #2       // not executed
done:
        b    done         // r2 is still 1
...
```

An ARM assembly language program for GNU

```
                .equ    STACK_TOP, 0x20000800
                .text
                .syntax unified
                .thumb
                .global _start
                .type   start, %function

_start:
                .word   STACK_TOP, start
start:
                movs r0, #10
                movs r1, #0
loop:
                adds r1, r0
                subs r0, #1
                bne  loop
deadloop:
                b    deadloop
                .end
```

A simple Makefile from https://github.com/embedded2013/arm-examples

```
all:
	arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
	arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
	arm-none-eabi-objcopy -Obinary example1.out example.bin
	arm-none-eabi-objdump -S example1.out > example1.list
```

An ARM assembly language program for GNU

```
            .equ    STACK_TOP, 0x20000800
            .text
            .syntax unified
            .thumb
            .global _start
            .type   start, %function

_start:
            .word   STACK_TOP, start
start:
            movs r0, #10
            movs r1, #0
loop:
            adds r1, r0
            subs r0, #1
            bne  loop
deadloop:
            b    deadloop
            .end
```

Disassembled object code

```
example1.out:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:       20000800                .word   0x20000800
   4:       00000009                .word   0x00000009

00000008 <start>:
   8:       200a      movs    r0, #10
   a:       2100      movs    r1, #0

0000000c <loop>:
   c:       1809      adds    r1, r1, r0
   e:       3801      subs    r0, #1
  10:       d1fc      bne.n   c <loop>

00000012 <deadloop>:
  12:       e7fe      b.n     12 <deadloop>
```

The endianess religious war: 286 years and counting!

- *Modern version*
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war

- *Historical Inspiration*
  - Jonathan Swift
  - *Gullivers Travels*
  - Published in 1726
  - Satire on Henry-VIII's split with the Church

- *Little-Endian*
  - LSB is at lower address

```
                                  Memory    Value
                                  Offset  (LSB) (MSB)
                                  ======  ===========
uint8_t a  = 1;                   0x0000  01 02 FF 00
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;          0x0004  78 56 34
12
```

- Big-Endian
  - MSB is at lower address

```
                                  Memory    Value
                                  Offset  (LSB) (MSB)
                                  ======  ===========
uint8_t a  = 1;                   0x0000  01 02 00 FF
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;          0x0004  12 34 56
78
```

Instruction encoding

- *Instructions are encoded in machine language opcodes*
- *Sometimes*
  - Necessary to hand generate opcodes
  - Necessary to verify assembled code is correct
- *How?*

Instructions

movs r0, #10

movs r1, #0

Register Value **Memory Value**

001|00|000|00001010 (LSB) (MSB)

(msb) (lsb) **0a 20 00 21**

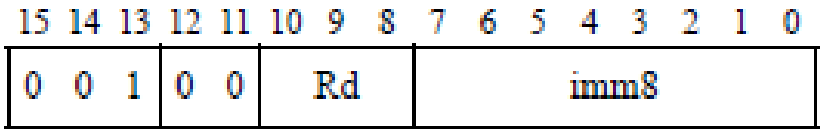
001|00|001|00000000

ARMv7 ARM

Encoding T1 All versions of the Thumb ISA.

MOVS <Rd>,#<imm8> Outside IT block.

MOV<c> <Rd>,#<imm8> Inside IT block.

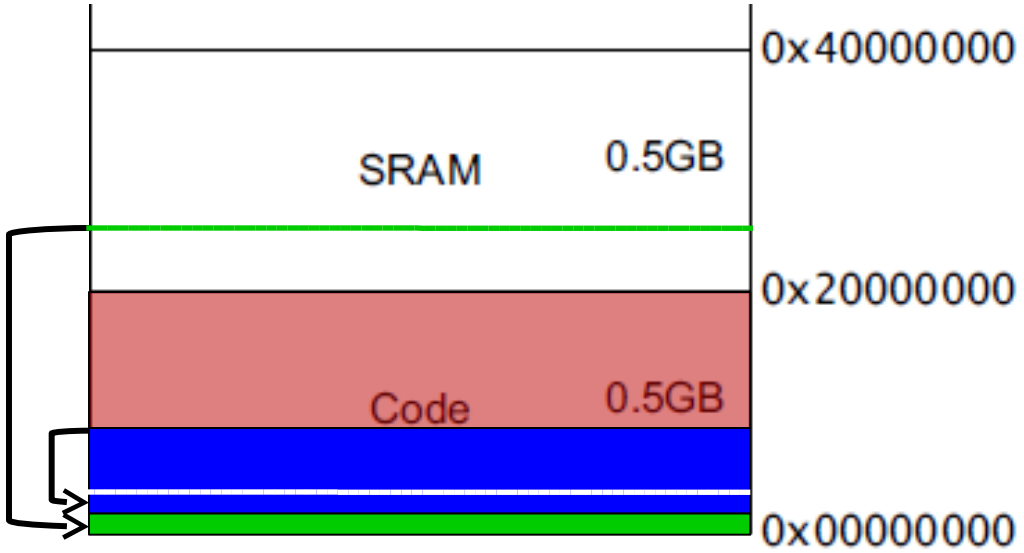



```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

What happens after a power-on-reset (POR)?

- *On the ARM Cortex-M3*
- *SP and PC are loaded from the code (.text) segment*
- *Initial stack pointer*
  - LOC: 0x00000000
  - POR: SP ← mem(0x00000000)
- *Interrupt vector table*
  - *Initial* base: 0x00000004
  - Vector table is relocatable
  - Entries: 32-bit values
  - Each entry is an address
  - Entry #1: reset vector
    - LOC: 0x0000004
    - POR: PC ← mem(0x00000004)
- *Execution begins*

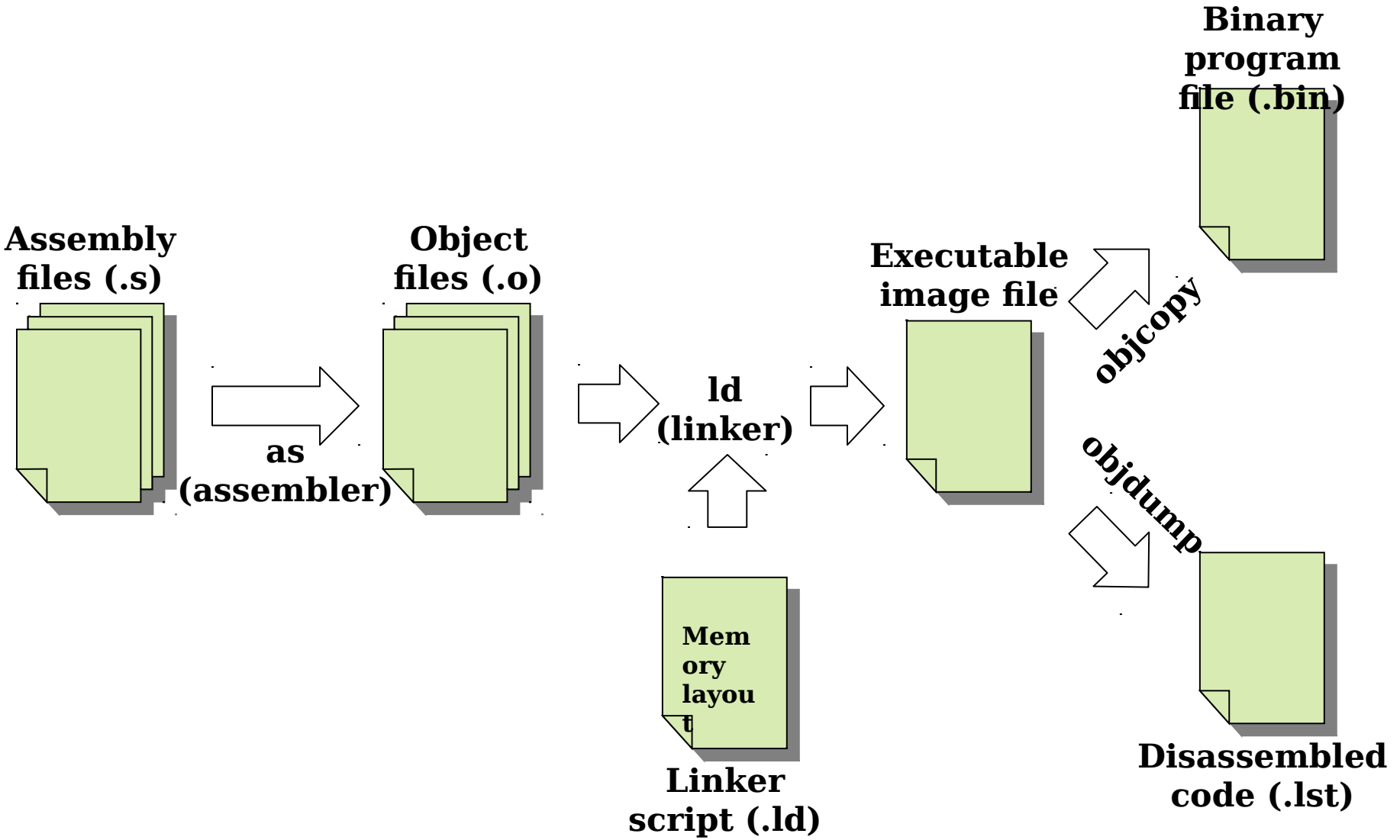
```
            .equ    STACK_TOP, 0x20000800
            .text
            .syntax unified
            .thumb
            .global _start
            .type   start, %function

_start:
            .word   STACK_TOP, start
start:
            movs r0, #10
            ...
```



Outline

- *Minute quiz*
- *Announcements*
- *ARM Cortex-M3 ISA*
- *Assembly tool flow*
- *C/Assembly mixed tool flow*

How does an assembly language program get turned into a executable program image?



What are the real GNU executable names for the ARM?

- *Just add the prefix "arm-none-eabi-" prefix*
- *Assembler (as):arm-none-eabi-as*
- *Linker (ld):arm-none-eabi-ld*
- *Object copy (objcopy): arm-none-eabi-objcopy*
- *Object dump (objdump): arm-none-eabi-objdump*
- *C Compiler (gcc): arm-none-eabi-gcc*
- *C++ Compiler (g++): arm-none-eabi-g++*

A simple (hardcoded) Makefile example

```
all:
	arm-none-eabi-as -mcpu=cortex-m3 -mthumb \
	   example1.s -o example1.o
	arm-none-eabi-ld -Ttext 0x0 \
	   -o example1.out example1.o
	arm-none-eabi-objcopy -Obinary \
	   example1.out example1.bin
	arm-none-eabi-objdump \
	   -S example1.out > example1.lst
```

What information does the disassembled file provide?

```
all:
        arm-none-eabi-as -mcpu=cortex-m3 -mthumb \
            example1.s -o example1.o
        arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
        arm-none-eabi-objcopy -Obinary \
            example1.out example1.bin
        arm-none-eabi-objdump -S example1.out > example1.lst
```

```
            .equ    STACK_TOP, 0x20000800
            .text
            .syntax unified
            .thumb
            .global _start
            .type   start, %function

_start:
            .word   STACK_TOP, start
start:
            movs r0, #10
            movs r1, #0
loop:
            adds r1, r0
            subs r0, #1
            bne  loop
deadloop:
            b    deadloop
            .end
```

```
example1.out:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   20000800        .word   0x20000800
   4:   00000009        .word   0x00000009

00000008 <start>:
   8:   200a            movs    r0, #10
   a:   2100            movs    r1, #0

0000000c <loop>:
   c:   1809            adds    r1, r1, r0
   e:   3801            subs    r0, #1
  10:   d1fc            bne.n   c <loop>

00000012 <deadloop>:
  12:   e7fe            b.n     12
<deadloop>
```

What are the elements of a real assembly program?

```
        .equ    STACK_TOP, 0x20000800    /* Equates symbol to value */
        .text                            /* Tells AS to assemble region */
        .syntax unified                  /* Means language is ARM UAL */
        .thumb                           /* Means ARM ISA is Thumb */
        .global _start                   /* .global exposes symbol */
                                         /* _start label is the beginning */
                                         /* ...of the program region */
        .type   start, %function                 /* Specifies start is a
function */
                                         /* start label is reset handler */
_start:
        .word   STACK_TOP, start                 /* Inserts word
0x20000800 */
                                         /* Inserts word (start) */
start:
        movs r0, #10                     /* We've seen the rest ... */
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```

Exercise

```
        .equ    STACK_TOP, 0x20000800   /* Equates symbol to value */
        .text                           /* Tells AS to assemble region */
        .syntax unified                 /* Means language is ARM UAL */
        .thumb                          /* Means ARM ISA is Thumb */
        .global _start                  /* .global exposes symbol */
                                        /* _start label is the beginning */
                                        /* ...of the program region */
        .type   start, %function                /* Specifies start is a
function */
                                        /* start label is reset handler */
_start:
        .word   STACK_TOP, start                /* Inserts word
0x20000800 */
                                        /* Inserts word (start) */
start:
        movs r0, #10                    /* We've seen the rest ... */
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```

How are assembly files assembled?

- *$ arm-none-eabi-as*
  - Useful options
    - -mcpu
    - -mthumb
    - -o

$ arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o

How can the contents of an object file be read?

- *$ readelf -a example.o*

- *Shows*
  - ELF headers
  - Program headers
  - Section headers
  - Symbol table
  - Files attributes

- *Other options*
  - -s shows symbols
  - -S shows section headers

What does an object file contain?

```
$ readelf -S example1.o
There are 9 section headers, starting at offset 0xac:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000014 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 000300 000008 08      7   1  4
  [ 3] .data             PROGBITS        00000000 000048 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000048 000000 00  WA  0   0  1
  [ 5] .ARM.attributes   ARM_ATTRIBUTES  00000000 000048 000021 00      0   0  1
  [ 6] .shstrtab         STRTAB          00000000 000069 000040 00      0   0  1
  [ 7] .symtab           SYMTAB          00000000 000214 0000c0 10      8  11  4
  [ 8] .strtab           STRTAB          00000000 0002d4 00002c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

How are object files linked?

- *$ arm-none-eabi-ld*
  - Useful options
    - -Ttext
    - -Tbss
    - -o

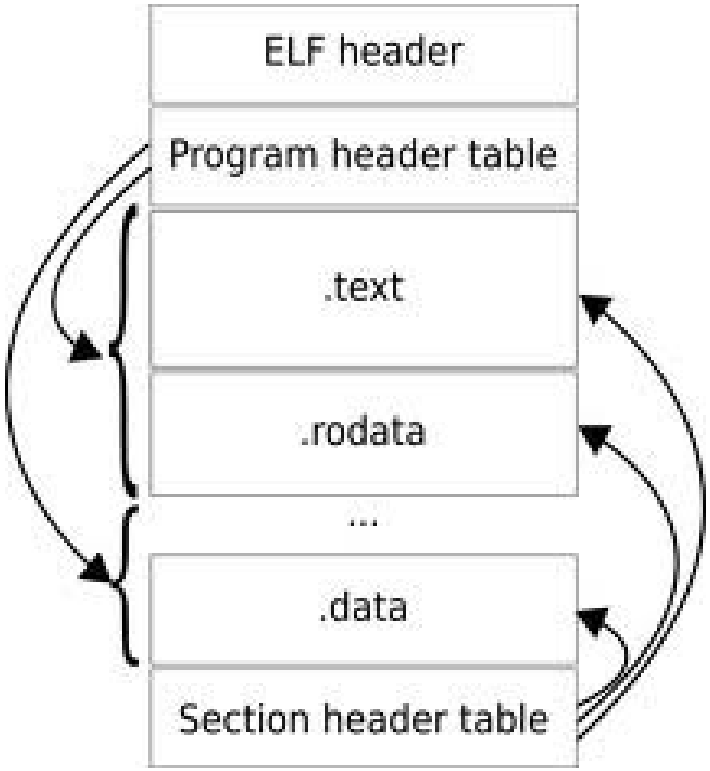
$ arm-none-eabi-ld -Ttext 0x0 –Tbss 0x20000000 -o example1.out example1.o

What are the contents of typical linker script?

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(main)

MEMORY
{
 ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

SECTIONS
{
 .text :
 {
   . = ALIGN(4);
    *(.text*)
            . = ALIGN(4);
    _etext = .;
 } >ram
}
end = .;
```

What does an executable image file contain?

- *.text segment*
  - Executable code
  - Initial reset vector
- *.data segment (.rodata in ELF)*
  - Static (initialized) variables
- *.bss segment*
  - Static (uninitialized) variables
  - Zero-filled by CRT or OS
  - From: Block Started by Symbol
- *Does not contain heap or stack*



- *For details, see:*

  /usr/include/linux/elf.h

How can the contents of an executable file be read?

- *Exactly the same way as an object file!*

- *Recall the useful options*
  - -a show all information
  - -s shows symbols
  - -S shows section headers

What does an executable file contain?

- *Use readelf's -S option*
- *Note that the .out has fewer sections than the .o file*
  - Why?

```
$ readelf -S example1.out
There are 6 section headers, starting at offset 0x8068:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 008000 000014 00  AX  0   0  4
  [ 2] .ARM.attributes   ARM_ATTRIBUTES  00000000 008014 000021 00      0   0  1
  [ 3] .shstrtab         STRTAB          00000000 008035 000031 00      0   0  1
  [ 4] .symtab           SYMTAB          00000000 008158 000130 10      5   9  4
  [ 5] .strtab           STRTAB          00000000 008288 000085 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

What are the contents of an executable's .text segment?

- *Use readelf's -x option to hex dump a section*
- *1st column shows memory address*
- *2nd through 5th columns show data*
- *The initial SP and PC values are visible*
- *The executable opcodes are also visible*

```
$ readelf -x .text example1.out

Hex dump of section '.text':
  0x00000000 | 00080020 09000000 0a200021 09180138 ... ..... .!...8
  0x00000010 | fcd1fee7                           ....
```

What are the raw contents of an executable file?

- *Use hexdump*
- *ELF's magic number is visible*
- *The initial SP, PC, executable opcodes are visible*

```
$ hexdump example1.out
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000010 0002 0028 0001 0000 0000 0000 0034 0000
0000020 8068 0000 0000 0500 0034 0020 0001 0028
0000030 0006 0003 0001 0000 8000 0000 0000 0000
0000040 0000 0000 0014 0000 0014 0000 0005 0000
0000050 8000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0000 0000 0000 0000
*
0008000 0800 2000 0009 0000 200a 2100 1809 3801
0008010 d1fc e7fe 2041 0000 6100 6165 6962 0100
0008020 0016 0000 4305 524f 4554 2d58 334d 0600
0008030 070a 094d 0002 732e 6d79 6174 0062 732e
0008040 7274 6174 0062 732e 7368 7274 6174 0062
0008050 742e 7865 0074 412e 4d52 612e 7474 6972
0008060 7562 6574 0073 0000 0000 0000 0000 0000
0008070 0000 0000 0000 0000 0000 0000 0000 0000
*
0008090 001b 0000 0001 0000 0006 0000 0000 0000
```

What purpose does an executable file serve?

- *Serves as a convenient container for sections/segments*
- *Keeps segments segregate by type and access rights*
- *Serves as a program "image" for operating systems*
- *Allows the loader to place segments into main memory*
- *Can integrates symbol table and debugging information*

How useful is an executable image for most embedded systems & tools?

What does a binary program image contain?

- *Basically, a binary copy of program's .text section*
- *Try 'hexdump -C example.bin'*
- *Want to change the program?*
  - Try 'hexedit example.bin'
  - You can change the program (e.g. opcodes, static data, etc.)
- *The initial SP, PC, executable opcodes are visible*

```
$ hexdump -C example.bin
00000000  00 08 00 20 09 00 00 00  0a 20 00 21 09 18 01 38  |....|
00000010  fc d1 fe e7                                       |....|
00000014

$ hexedit example.bin
00000000   00 08 00 20  09 00 00 00  0A 20 00 21  09 18 01 38  ....
00000010   FC D1 FE E7                                  ....
```

What are other, more usable formats?

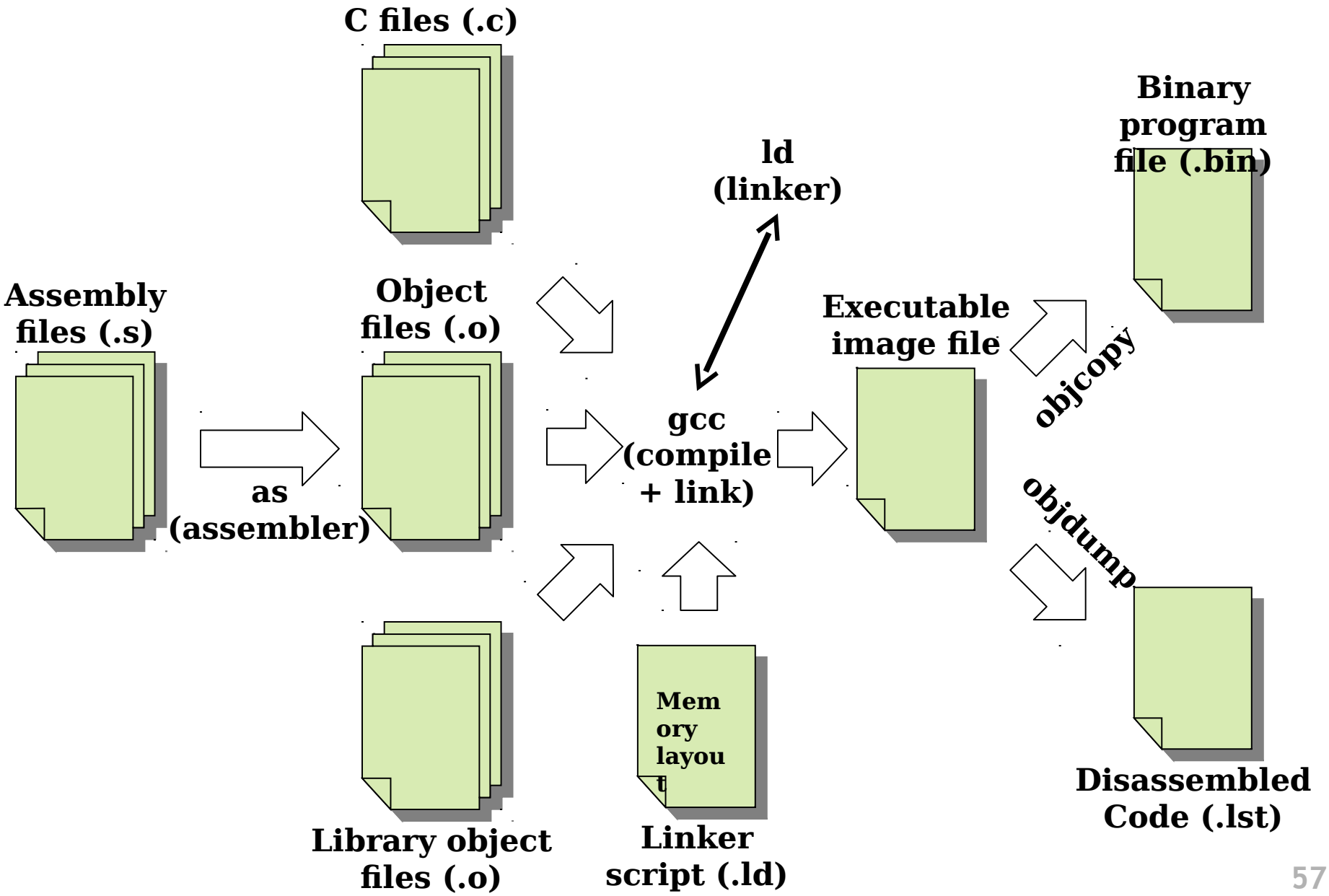
- *.o, .out, and .bin are all binary formats*
- *Many embedded tools don't use binary formats*
- *Two common ASCII formats*
  - Intel hex (ihex)
  - Motorola S-records (srec)
- *The initial SP, PC, executable opcodes are visible*

```
$ arm-none-eabi-objcopy -O ihex example1.out "example1.hex“
$ cat example1.hex
:10000000000800200900000000A2000210918013811A
:04001000FCD1FEE73A
:00000001FF

$ arm-none-eabi-objcopy -O srec example1.out "example1.srec“
$ cat example1.srec
S0100000657861D706C65312E73726563F7
S113000000080020090000000A2000210918013816
S1070010FCD1FEE736
S9030000FC
```

Outline

- *Minute quiz*
- *Announcements*
- *ARM Cortex-M3 ISA*
- *Assembly tool flow*
- *C/Assembly mixed tool flow*

How does a mixed C/Assembly program get turned into a executable program image?



Cheap trick: use asm() or __asm() macros to sprinkle simple assembly in standard C code!

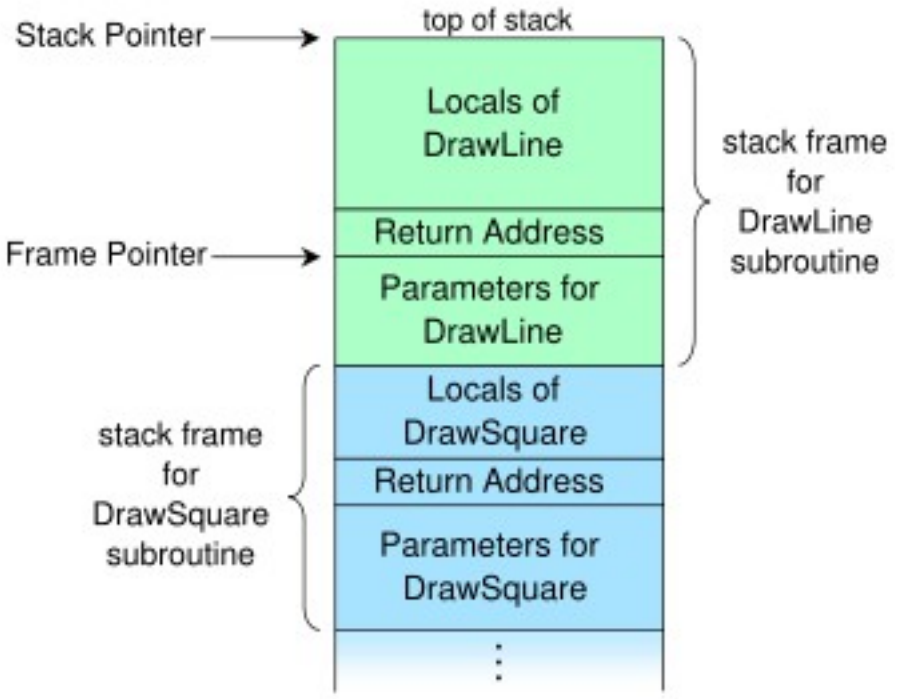
```
int main() {
  int i;
  int n;
  unsigned int input = 40, output = 0;
  for (i = 0; i < 10; ++i) {
    n = factorial(i);
    printf("factorial(%d) = %d\n", i, n);
  }
  __asm("nop\n");
  __asm("mov  r0, %0\n"
        "mov  r3, #5\n"
        "udiv r0, r0, r3\n"
        "mov  %1, r0\n"
        :"=r" (output)
        : "r" (input)
        : "cc", "r3" );
  __asm("nop\n");
  printf("%d\n", output);
}
```

Answer: 40/5 →

```
$ arm-none-eabi-gcc \
  -mcpu=cortex-m3 \
  -mthumb main.c \
  -T generic-hosted.ld \
  -o factorial
$ qemu-arm -cpu
cortex-m3 \
  ./factorial
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
factorial(9) = 362880
8
```

Passing parameters via the stack

- *Benefits?*
- *Drawbacks?*



Passing parameters via the registers/stack

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

ABI Basic Rules

1. *A subroutine must preserve the contents of the registers r4-r11 and SP*

2. *Arguments are passed though r0 to r3*
   - If we need more, we put a pointer into memory in one of the registers.
     - We'll worry about that later.

3. *Return value is placed in r0*
   - r0 and r1 if 64-bits.

4. *Allocate space on stack as needed. Use it as needed.*
   - Put it back when done…
   - Keep word aligned.

Other useful facts

- *Stack grows down.*
  - And pointed to by “SP”
- *Address we need to go back to in “LR”*

And useful things for the example

- *Assembly instructions*
  - add adds two values
  - mul multiplies two values
  - bx branch to register

A simple ABI routine

- *int bob(int a, int b)*
  - returns $a^2 + b^2$
- *Instructions you might need*
  - add adds two values
  - mul multiplies two values
  - bx branch to register

Outline

- *Minute quiz*
- *Announcements*
- *Review*
- *Assembly, C, and the ABI*
- *Memory*
- *Memory-mapped I/O*

Memory-mapped I/O

- *The idea is really simple*
  - Instead of real memory at a given memory address, have an I/O device respond.
- *Example:*
  - Let's say we want to have an LED turn on if we write a "1" to memory location 5.
  - Further, let's have a button we can read (pushed or unpushed) by reading address 4.
    - If pushed, it returns a 1.
    - If not pushed, it returns a 0.

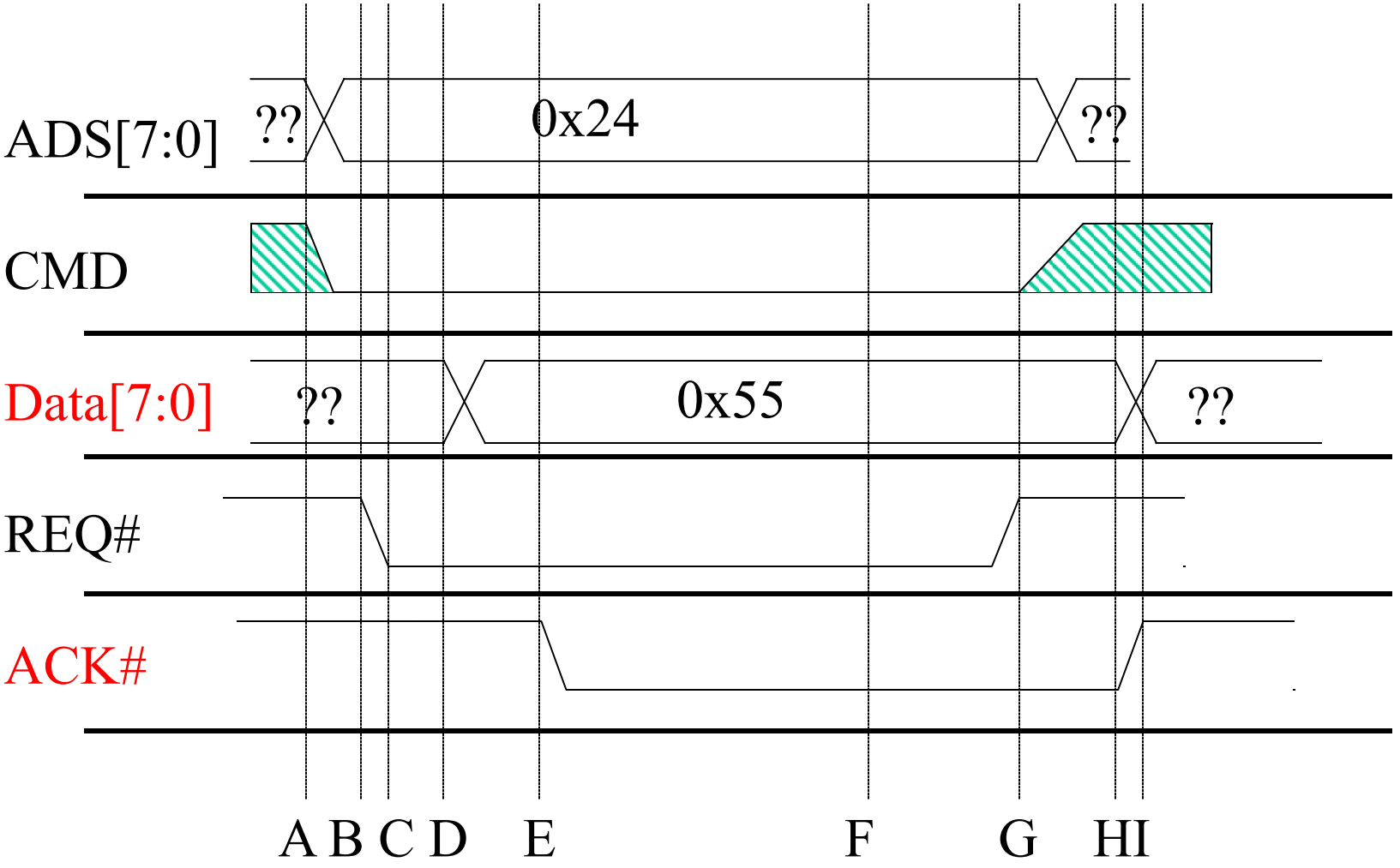
Basic example

- *Discuss a basic bus protocol*
  - Asynchronous (no clock)
  - Initiator and Target
  - REQ#, ACK#, Data[7:0], ADS[7:0], CMD
    - CMD=0 is read, CMD=1 is write.
    - REQ# low means initiator is requesting something.
    - ACK# low means target has done its job.

A read transaction

- *Say initiator wants to read location 0x24*
  - Initiator sets ADS=0x24, CMD=0.
  - Initiator *then* sets REQ# to low. (why do we need a delay? How much of a delay?)
  - Target sees read request.
  - Target drives data onto data bus.
  - Target *then* sets ACK# to low.
  - Initiator grabs the data from the data bus.
  - Initiator sets REQ# to high, stops driving ADS and CMD
  - Target stops driving data, sets ACK# to high terminating the transaction

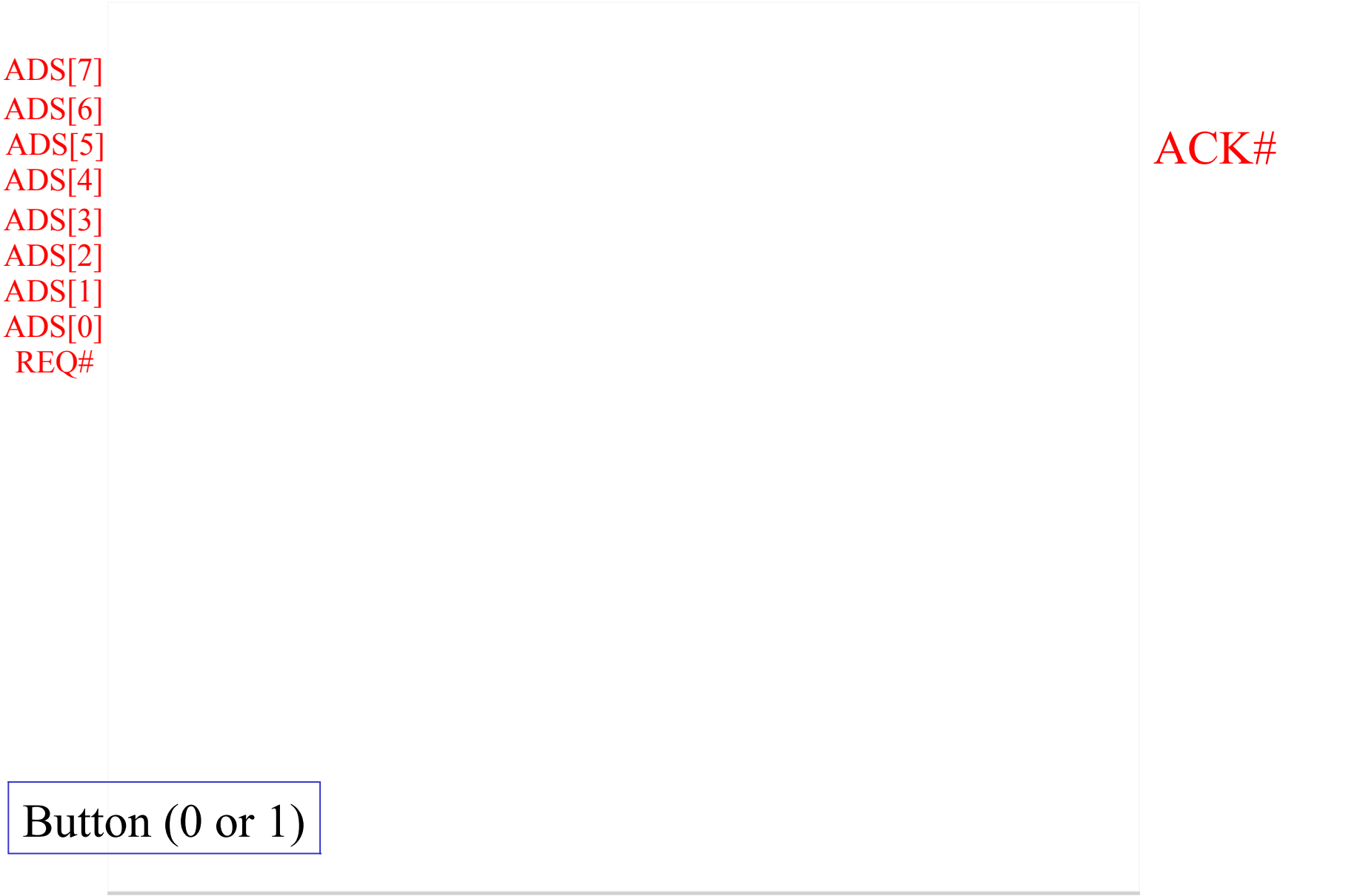
Read transaction



A write transaction
(write 0xF4 to location 0x31)

- Initiator sets ADS=0x31, CMD=1, Data=0xF4
- Initiator *then* sets REQ# to low.
- Target sees write request.
- Target reads data from data bus. (Just has to store in a register, need not write all the way to memory!)
- Target *then* sets ACK# to low.
- Initiator sets REQ# to high & stops driving other lines.
- Target sets ACK# to high terminating the transaction

The push-button
(if ADS=0x04 write 0 or 1 depending on button)



The push-button
(if ADS=0x04 write 0 or 1 depending on button)